Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

TRABAJO DE FIN DE GRADO

DESARROLLO DE UN GENERADOR PROCEDURAL DE TERRENOS

Miguel Gargallo Vázquez
Tutor: Carlos Aguirre Maeso

3 de julio de 2015

DESARROLLO DE UN GENERADOR PROCEDURAL DE TERRENOS

Autor: Miguel Gargallo Vázquez
Tutor: Carlos Aguirre Maeso

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

3 de julio de 2015

Abstract

Abstract — Procedural content generation is getting more and more relevant in areas such as level generation in video games, texture and particle generation in animation and visual effects...

With development teams getting smaller in video game development, making successful titles in today's market when it comes to offering enough play hours, against big development companies, is impossible without using techniques that will save the team the effort of crafting the levels by hand. Here is where procedural generation engines come into play.

This assignment aims to develop a procedural generation tool for terrains. It will be very customizable and simple to use, and it will be built for the web, thus allowing development teams to forget about the algorithms behind procedural generation and focus on their game.

After deciding the goal for the product's first version and establishing all the essential details of the development, such as the appropiate algorithms and data structures, a functional prototipe is obtained, integrating some basic techniques of terrain generation.

When deciding which algorithms should be implemented in this prototipe, we tried to keep development as simple and clear as possible. Even with this approach, the results were good and the performance was excelent.

However, there were also some problems derived from following this approach of keeping it as simple as possible. These made the generated terrains feel less natural, or do not allow all the flexibility we were looking for. A good example is river generation, which is the most complex module of the project, and whose simplification brought errors common in many situations, and for example prevent the user from deciding not to generate oceans.

The final thought is that the results obtained were good despite all the problems that all the simplifications brought, and, on the other hand, these simplifications have allowed us to implement a bigger number of modules than we had been able to if we had chosen to follow a more realistic and complex approach.

Key words — WebGL, Procedural, 2D, 3D

Resumen

Resumen — La generación procedural de contenido toma cada vez más relevancia en campos como la generación de niveles en videojuegos, generación de texturas y partículas en animación y efectos especiales...

Con equipos de desarrollo cada vez más pequeños en el mundo de los videojuegos, crear títulos competentes en el mercado actual a la hora de ofrecer horas de juego al usuario frente a juegos de grandes compañías es imposible si no se emplean técnicas que ahorren al equipo la creación de niveles a mano. Ahí es donde entran en juego los motores de generación procedural.

Este trabajo pretende crear una herramienta de generación procedural de terrenos muy personalizable y simple de usar para la plataforma web, permitiendo a equipos de desarrollo abstraerse de los algoritmos de generación para centrarse en la creación de su juego.

Tras decidir el alcance de la primera versión del producto y establecer los detalles cruciales del desarrollo del mismo, como los algoritmos y estructuras de datos más apropiados, se obtiene un prototipo funcional que integra algunas técnicas básicas de generación de terrenos.

A la hora de decidir qué algoritmos debían implementarse en este prototipo, se ha seguido el enfoque de simplificar y clarificar lo máximo posible el desarrollo del producto. Aun así se han conseguido resultados muy aceptables y con un rendimiento excelente.

Sin embargo, también se han encontrado problemas derivados de la simplificación, que hacen que los terrenos generados pierdan naturalidad y parezcan más artificiales, o que impiden obtener toda la flexibilidad que se busca. Un ejemplo es la generación de ríos, que constituye el módulo más complejo del proyecto, y cuya simplificación provoca imperfecciones comunes en muchas situaciones, e impiden que por ejemplo se decida no generar océanos en un mapa.

La conclusión final es que los resultados son buenos a pesar de los problemas que acarrean las simplificaciones que se han llevado a cabo, y éstas han permitido por otro lado implementar un mayor número de módulos de los que habrían sido posibles si se hubiera intentado seguir un enfoque más realista y complejo.

Palabras clave — WebGL, Procedural, 2D, 3D

Glosario

- **Mapa de alturas** Imagen en 2 dimensiones que, a través del color de cada uno de sus píxeles, dan el valor de altura (tercera dimensión) de cada uno de los elementos que representa.
- **Mapa topográfico** Representación terrestre que, a través de líneas y colores, indica la altura de un terreno con unas franjas de alturas del mismo color llamadas çurvas de nivel".
- **Procedural** Generado a través de un algoritmo.
- **Renderizar** Generar una imagen a partir de un conjunto de datos.
- **Ruido** En este contexto, un conjunto aleatorio de datos.

Índice general

Índice de tablas

Índice de figuras

1

Introducción

1.1. Motivación

La generación procedural es una técnica cada vez más empleada en los últimos años, tanto en lo que se refiere a la generación de contenido en sí mismo, como puede ser el caso de la generación de niveles aleatorios en los videojuegos, como la generación de texturas para simular superficies y materiales, una técnica ampliamente usada tanto en el campo de los videojuegos como en el de la animación o incluso efectos especiales.

A pesar de sus amplias aplicaciones, es muy destacable su uso en el campo de los videojuegos, donde se viene empleando desde los inicios de los años 80 con videojuegos como *Rogue* [1], título que ha acuñado el término *roguelike* (literalmente, çomo Rogue"). El *roguelike*[2] es un subgénero del RPG en el se genera proceduralmente una mazmorra al comienzo de cada partida, en la cual el jugador puede progresar hasta que su personaje muere. En ese momento, la partida finaliza y una nueva ha de ser comenzada para continuar jugando. Éste es un ejemplo donde la generación procedural de contenido juega un papel primordial en la experiencia de juego.

[1] https://en.wikipedia.org/wiki/Rogue_(video_game)
[2] https://en.wikipedia.org/wiki/Roguelike

En los últimos años esta técnica ha sufrido de nuevo un gran auge, debido a que el abaratamiento de los costes de desarrollo y distribución permiten a equipos de personas cada vez más pequeños e independientes desarrollar sus propios videojuegos.

A raíz de esto, resulta imposible para un equipo de unas pocas personas hacer frente a equipos de grandes empresas a la hora de llenar su videojuego de contenido y ofrecer horas de juego al usuario. Por ello, muchos equipos optan por la generación procedural de contenido, de tal forma que, desarrollados los algoritmos (que no es en ningún caso una tarea simple), el mundo se regenera en cada partida, haciendo que cada una dé la impresión de ser fresca, y ampliando así las horas posibles de juego.

Los ejemplos son incontables: Minecraft [3] fue desarrollado en sus primeras etapas por una única persona, Terraria [4] por un equipo de 3 personas, y Faster Than Light [5] (un ejemplo de un roguelike moderno con premios internacionales) por dos personas. Los tres son conocidos por la enorme cantidad de horas de juego que ofrecen (fácilmente superior a las 100), y esto sería obviamente imposible de crear, no ya sólo por un equipo tan pequeño, sino por una gran empresa de desarrollo, sin las técnicas de generación procedural.

Este proyecto pretenderá aportar herramientas de desarrollo para una plataforma web para el programador que desee crear un videojuego que emplee las técnicas de generación procedural de contenidos, pudiéndose abstraer de éstas, para así facilitar y acelerar el desarrollo del videojuego en sí mismo.

1.2. Objetivos

El trabajo que se está presentando emplea ciertas técnicas básicas de generación procedural de contenidos para tratar de generar terrenos de manera muy personalizada.

Se presentará al usuario en forma de biblioteca de desarrollo (pues no se pretende crear un producto final), debidamente organizada y documentada para que su uso sea muy sencillo, y se separarán cada una de las capacidades para dar al usuario esa personalización y flexibilidad que se busca.

Para este trabajo, dicha biblioteca supondrá un primer prototipo en el proyecto, dejando ver lo que será el esqueleto del producto final e implementando las primeras funcionalidades de generación, como es la generación de un mapa de alturas y la generación de agua.

[3] https://minecraft.net/
[4] https://terraria.org/
[5] www.ftlgame.com/

También se tratará de implementar un pequeño motor de renderizado que aportar al usuario, aunque su objetivo es mayormente el de ser una herramienta de depuración y pruebas, no el de ser parte del producto final.

1.3. Estructura del documento

Tras esta pequeña introducción al tema y un posterior análisis de la situación actual de herramientas como ésta en el mercado, se comienza a hablar de la creación de dicha herramienta, pasando por las diferentes etapas que han tenido lugar.

Durante la etapa de diseño que se presenta a continuación se ha analizado hasta qué punto se podría implementar esta herramienta, ya que, partiendo de un cierto esqueleto, añadir progresivas funcionalidades es muy sencillo e iterativo. Habiendo decidido qué se puede conseguir, se comentan las decisiones que se han tomado para la elección de los algoritmos que se han empleado finalmente en cada uno de los módulos, y de las estructuras de datos necesarias para llevar estas tareas a cabo.

Después se describe la etapa de desarrollo, que ha consistido en la implementación de los módulos anteriormente planteados. También se discuten ciertos aspectos de programación que han sido relevantes a lo largo del desarrollo, y se explican los algoritmos elegidos, cada una de sus ventajas e inconvenientes, y el enfoque tomado durante el trabajo.

Por último, en el capítulo de resultados, se comentan las pruebas que se han ido llevando a cabo durante todo el desarrollo del producto y las plataformas que se han montado para facilitar las mismas, así como las interpretaciones de los resultados obtenidos. También se comentan ciertas dificultades y problemas que se han encontrado a lo largo del proyecto debido al enfoque que se ha dado a ciertos módulos al intentar simplificar las tareas.

2

Estado del Arte

2.1. Generadores procedurales de terrenos

La generación procedural de contenido para videojuegos no es un tema nuevo; muchos desarrolladores llevan empleando estas técnicas para aportar variedad a sus videojuegos durante décadas. Pero el número de motores de generación que se han ofrecido o liberado al público no es tan grande, ya que la mayoría constituyen herramientas internas del equipo de desarrollo.

Además, si se busca trabajar para una plataforma en concreto, las posibilidades se pueden reducir más todavía. En el caso de la web, que muchos preveen será la plataforma más relevante de los próximos años, el abanico de herramientas encontrado es aún muy reducido, sobre todo si se busca una herramienta gratuita.

Motores multiplataforma como Unity llevan incorporadas herramientas de generación de terrenos muy completas, que muchas empresas utilizan. El problema con estas herramientas es la customización. Al ser código cerrado y formar parte de una plataforma comercial, los desarrolladores quedan de cierta forma .ª la merced"de lo que la herramienta les ofrece. Esto puede ser aceptable si el resultado que se busca es realista, ya que es lo que tienden a hacer estas herramientas, pero si se busca un resultado distinto por motivos

artísticos, el diseñador queda restringido por la herramienta.

No se han encontrado herramientas abiertas para web lo suficientemente potentes como para ser empleadas en el desarrollo de un videojuego. La mayoría de las existentes sólo se hicieron a modo de demostración del rendimiento de WebGL renderizando terreno con iluminación, y no tienen el propósito de crear algo que pueda servir para posteriores desarrollos.

3

Diseño

Como se ha venido diciendo hasta ahora, el objetivo del trabajo es generar proceduralmente terrenos naturales; no obstante, una parte importante del desarrollo era la visualización de los resultados obtenidos. Es por ello que el proyecto desarrollado puede ser dividido en dos partes: generación y renderizado. Esto aporta al usuario además una funcionalidad de renderizado básica que él puede o no sobreescribir y, por tanto, utilizar.

La idea del proyecto es aportar una interfaz al usuario en forma de archivo de javascript, con la funcionalidad de generación encapsulada en una serie de funciones y constantes que permiten generar un terreno personalizado de manera muy sencilla.

La generación sigue una serie de pasos en forma de llamadas a funciones que han de ser hechas explícitamente por el usuario, de tal forma que éste puede decidir saltarse un paso o elegir entre distintos algoritmos para cada uno de los pasos.

Un ejemplo, en el que cada paso consiste simplemente en dar valor a una serie de constantes y luego realizar la correspondiente llamada a función, sería:

1. **Generar un mapa de alturas**. Establecer el tamaño y la rugosidad deseados para el mapa, y hacer la llamada a la función de generación de ruido Perlin.

2. **Generar océanos**. Establecer el nivel del mar y hacer la llamada al algoritmo de

generación de océanos deseado.

3. **Generar ríos**. Indicar la frecuencia de ríos deseada y llamar a la función de generación de ríos por A*.

4. **Generar vegetación**. Llamar a la función de generación de vegetación en función del clima.

5. **Renderizado**. Indicar el nombre del elemento canvas de salida y llamar a la función de renderizado en 3D con Voxels.

El código está estructurado de forma completamente modularizada, con el propósito de que ampliar las funcionalidades en el futuro sea lo más simple posible.

Por ejemplo, añadir un algoritmo de generación de ruido nuevo consistirá únicamente en implementar la función concreta (aunque puede emplear funciones secundarias, por supuesto) y crear las constantes necesarias. Después, lo único que el usuario tendría que hacer si quiere usarlo es dar valores a las nuevas constantes y llamar a la nueva función en el paso de la generación del mapa de alturas.

3.1. Prototipo básico

El objetivo esencial de este trabajo era conseguir diseñar e implementar un prototipo básico con los primeros módulos y donde se pudiera ver el esqueleto y el funcionamiento planteado del proyecto. Estos módulos incluyen las siguientes funcionalidades:

1. Generación de un mapa de alturas empleando el algoritmo de generación de ruido Diamond-Square. Esta funcionalidad es básica y necesaria para el resto, pues se basan en este mapa.

2. Generación de océanos.

3. Generación de ríos.

4. Renderizado en 2D y 3D.

3.2. Estructuras de datos

Existen dos estructuras de datos básicas en esta primera iteración:

- **Elemento** Constituye el elemento básico del que se compone el terreno generado. Contiene dos atributos: el material, que puede ser tierra o agua en esta versión; y la altura.

- **Tipos de terreno** Una enumeración de los tipos de terreno que distinguen los algoritmos de renderizado en función de varias características del terreno. En esta primera versión, sólo se tienen en cuenta el material del terreno y la altura, y se distinguen los siguientes tipos de terreno, ordenados de mayor profundidad a menor profundidad, e indicando el color asociado a cada tipo:

  1. Agua
     a) Agua profunda: #0E0E3A (azul oscuro)
     b) Agua superficial: #369A7C (turquesa)
     c) Río: #5279BB (azul cielo)
  2. Tierra
     a) Costa: #CCC479 (amarillo)
     b) Llanura: #709A40 (verde claro)
     c) Bosque: #39712D (verde oscuro)
     d) Colinas: #955C3B (marrón)
     e) Montaña: #FFFFFF (blanco)

  Este color es el que será utilizado posteriormente por los algoritmos de renderizado, aunque en posteriores versiones se podría emplear otro parámetro como una serie de texturas para cada tipo de terreno. La estructura de datos admite un número variable de parámetros gracias a la flexibilidad de javascript.

3.3. Generación de ruido con Diamond-Square

La generación de mapa de alturas utiliza el algoritmo de generación de ruido Diamond-Square. Se ha elegido por su enorme simpleza y resultados aceptables en términos de eficiencia y requisitos de memoria.

Este último aspecto es importante, ya que la memoria disponible al programar en un navegador web es generalmente escasa, y en un caso como éste en el que hay que mantener grandes cantidades de datos almacenados (arrays de millones de elementos) y la dificultad para emplear métodos de almacenamiento alternativo (como escritura en archivos en disco de datos que no se vayan a emplear inmediatamente).

Los resultados gráficos de Diamond-Square son aceptables. Para ciertas rugosidades y tamaños pueden observarse patrones diagonales en forma de ondas debidos al proceso de subdivisión del terreno, pero dependiendo del algoritmo de renderizado desarrollado pueden ser imperceptibles.

3.4. Generación de agua

La generación de agua no es totalmente esencial en nuestro proyecto (el usuario podría saltarse este paso si desea paisajes sin agua), pero en la gran mayoría de casos será una fase importante que modificará tremendamente el terreno.

Este paso será desglosado en distintas subfases para dar mayor flexibilidad al usuario. Para el prototipo inicial, sólo se implementarán las fases de generación de océanos y la de generación de ríos, pero otras fases podrían ser la generación de lagos, pantanos, hielo...

3.5. Renderizado en 2D

La versión básica del renderizado en 2D permitirá visualizar el terreno generado como si se tratara de un mapa topográfico, donde las curvas de nivel vienen representadas por cambios en el color del terreno, de manera que se pueden distinguir de forma básica diferentes climas asociados con la altitud. No existe el concepto de latitud en esta primera versión.

El renderizado topográfico utiliza el enumerado de tipos de terrenos para obtener el color asociado a cada altura, y colorea los elementos que se encuentren en la franja definida por esa altura con ese color. Así se forman las curvas de nivel características de los mapas topográficos.

También se permite otro tipo de renderizado en escala de grises, aunque no está pensado para un uso final sino para que sirva como herramienta de depuración para la fase de generación de ruido, ya que permite observar claramente si se han generado patrones, así como la calidad del ruido.

3.6. Renderizado en 3D

El renderizado en 2D es sencillo, y por eso se ha hecho directamente en el código, pero el renderizado en 3D añade mucha complejidad. Es por esto que se ha empleado una biblioteca externa sobre WebGL que encapsule algunas tareas básicas y estructuras de datos con las que trabajar de manera más rápida, en vez de trabajar directamente sobre WebGL.

Por su simpleza de uso y su potencia, se ha elegido three.js.

3.6.1. three.js

three.js es una biblioteca de javascript desarrollada por Ricardo Cabello (conocido como Mr.doob) que actualmente goza de una comunidad muy activa de usuarios y desarrolladores en su proyecto de Github.

En la sección de ejemplos de la página web del proyecto (threejs.org) muestra el potencial del mismo con algunas páginas creadas por terceros que emplean three.js. Ahí se pueden ver desde simulaciones físicas como tejidos hasta renderizados realistas de piel humana.

Para el primer prototipo, se quiere proporcionar una interfaz para renderizado en 3D con una única llamada, de manera que se comporte igual que el renderizado en 2D. En siguientes prototipos, se podrían aportar distintas técnicas de renderizado en forma de funciones para que el usuario tenga más libertad a la hora de elegir una técnica de renderizado sin tener que elaborar la suya propia.

Algunas de las técnicas de renderizado que serían adecuadas para este proyecto son:

- **Voxels**. Un voxel es la correspondencia a un pixel en tres dimensiones (el nombre procede de "volume pixel", "pixel con volumen"). En este caso, cada elemento se ve representado por un cubo. Un famoso ejemplo de esta técnica de renderizado de terreno es el famoso videojuego Minecraft.

  Existen numerosos videojuegos del género *Sandbox*, donde el usuario puede modificar el mundo con libertad, que siguen la estela de Minecraft y emplean motores de voxels, aunque finalmente es una regresión a la construcción con bloques de Lego. En la figura 3.1 podemos ver un ejemplo de un cuerpo 3D construido a partir de voxels; en este caso, una seta amanita.

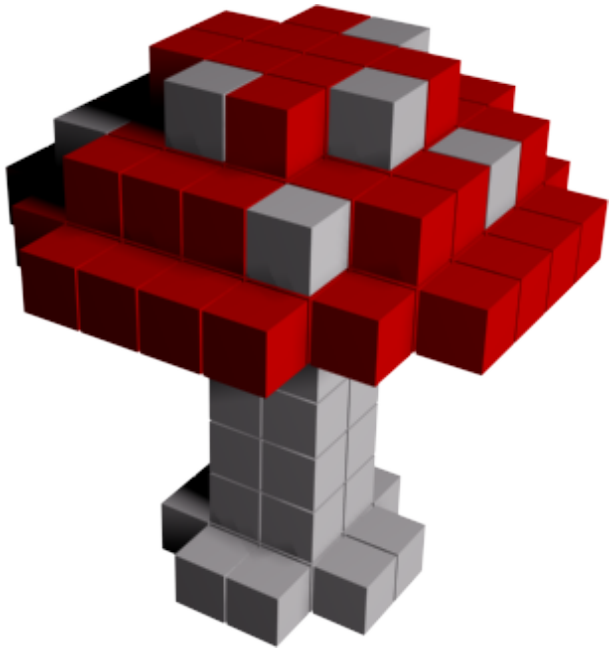

Figura 3.1: Representación de una seta formada a partir de voxels

- **Nube de puntos**. Una nube de puntos o *point cloud* es, como su propio nombre indica, una técnica de renderizado en la cual una serie de "puntos", correspondientes a los datos que se quieren representar, "flotan"definiendo el cuerpo. En la figura 3.2 se puede ver un ejemplo de una nube de puntos representando un toro.

  Estas nubes varían en densidad y pueden ser desde una representación esquemática de un cuerpo que acaba de ser escaneado hasta una imagen de gran resolución con calidad fotográfica, puesto que, correctamente orientados, los puntos son al fin y al cabo píxeles visibles desde distintos ángulos.

- **Malla de polígonos**. Si se unen los elementos a través de un algoritmo que cree caras poligonales, se obtiene una malla que da una representación con interpolación del terreno.

  Por ejemplo, se pueden tomar puntos por triples y crear triángulos. Esta técnica es conocida como triangulación.

  Esto permite de cierta forma ampliar la resolución del mismo, pues se obtienen datos que antes no existían entre dos puntos a la hora de renderizar. Por otro lado, se está modificando la forma del terreno, pero esto es algo que ocurre de forma más exagerada con la representación por voxels.

  Esta técnica tiene dos problemas de rendimiento: traducir el conjunto de elementos en una malla de polígonos no es excesivamente sencillo de implementar, y puede tener un coste computacional considerable; y representar una malla de polígonos

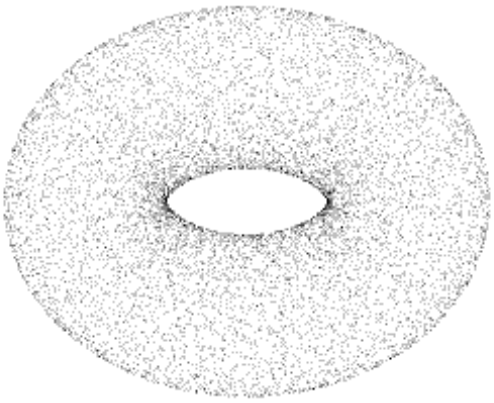

Figura 3.2: Un toro renderizado como nube de puntos

que no han sido optimizados es mucho más costoso que representar una nube de puntos en forma de cuadrados o un montón de cubos, pues son conjuntos de un único tipo de geometría y una misma orientación. Los polígonos, aunque sean de un mismo tipo, están deformados constantemente por la lejanía de los elementos entre sí.

En la figura 3.3 se puede observar un ejemplo de una malla poligonal formada por triángulos.

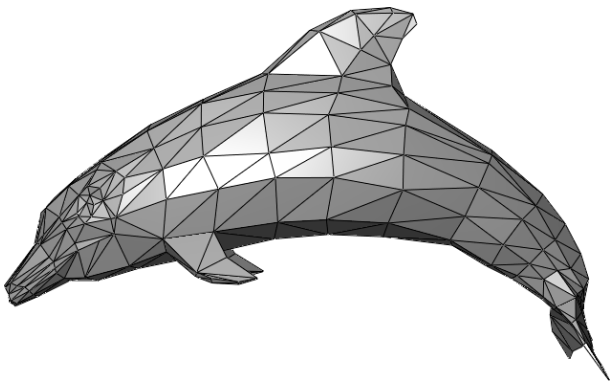

Figura 3.3: Malla de polígonos formada por triángulos

Una forma de mejorar este rendimiento es optimizar las estructuras de datos de la malla poligonal. Por ejemplo, un conjunto de triángulos pueden ser optimizados para formar lo que se conoce como "tiras de triángulos"(*triangle stripes*) o .ªbanicos de triángulos"(*triangle fans*), que permiten pasar a la tarjeta gráfica grupos de

triángulos de una forma más compacta. Esto se debe a que cuando se agrupan así, muchos vértices se sobreentienden por ser redundantes y no es necesario mandarlos; por ejemplo, el vértice central es común a todos los triángulos en un abanico, y cada par de triángulos adyacentes comparte otro vértice más, por lo que, definido un triángulo, el único dato necesario para definir el siguiente es un único vértice.

En la figura 3.4 puede observarse un ejemplo de abanico de triángulos.

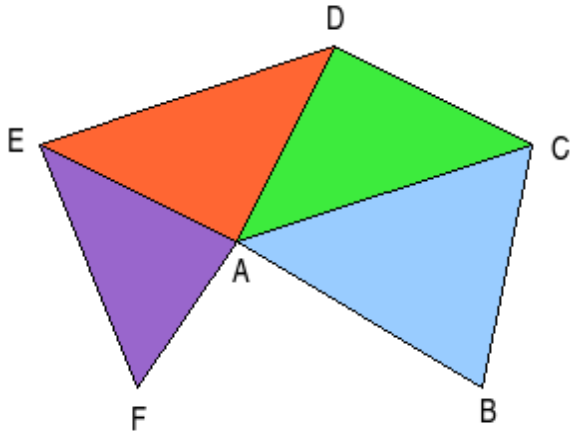

Figura 3.4: Abanico de triángulos

Por su simpleza de uso y el hecho de que no modifica el terreno (no se valora el realismo en este prototipo, pues es lo más costoso de conseguir y es hasta posible que no sea lo que busca el usuario), se va a emplear la técnica de renderizado 3D con nube de puntos para el primer prototipo del proyecto.

3.7. La página web

Para implementar todos los algoritmos anteriormente mencionados utilizaremos, como se viene diciendo, tecnología web. En concreto, javascript plano.

La biblioteca desarrollada se compondrá en el futuro de un único archivo de código javascript, pero para facilitar el desarrollo se va a separar, durante el mismo, cada uno de los módulos en un archivo distinto de código, de manera que tendremos los siguientes:

- **diamond_square.js** Implementará el algoritmo de generación de ruido Diamond-Square, que rellenará la estructura terrainData con los datos del ruido generado.

- **water.js** Implementará la generación de océanos y ríos. En la primera versión, implementará el algoritmo de generación por altura para los océanos y el de búsqueda del océano más cercano para los ríos, modificando la estructura terrainData.
- **render.js** Tendrá tres funciones que permitirán renderizar en escala de grises (para depurar la generación de ruido), en mapa topográfico y en point cloud.

Todos los algoritmos previamente mencionados serán discutidos en profundidad en la sección correspondiente del capítulo de desarrollo.

Cuando se desee exportar la biblioteca para su uso, sólo se tienen que concatenar los archivos en uno único, sin preocuparse del orden.

Será necesario ejecutar el código durante la fase de desarrollo para comprobar los resultados y permitir realizar pruebas de funcionamiento y rendimiento. Para ello, se creará una serie de archivos HTML con distintas funcionalidades, que se discutirán más adelante en el capítulo de desarrollo.

4

Desarrollo

El ciclo de vida elegido para el software ha sido el desarrollo por prototipos. Dada la modularidad y escalabilidad del proyecto, este proceso de desarrollo permitía partir de un prototipo básico consistente en la generación de un mapa de alturas y renderizado, para después poder añadir, en sucesivos prototipos, algoritmos que modificaran el terreno de distintas formas.

4.1. Prototipo básico

Como se ha mencionado anteriormente, el prototipo básico es el objetivo principal del trabajo. Para desarrollar cada uno de los módulos planteados, se han elegido algoritmos eficientes en tiempo de ejecución y simples de implementar, para poder abarcar cada una de las fases ideadas con el tiempo disponible.

Para la fase de generación no se han empleado herramientas externas (como librerías), sino que se han implementado los algoritmos y las estructuras de datos directamente en nuestro código javascript.

4.2. Implementación de Diamond-Square

Diamond-Square es un algoritmo de lo que se conoce como generación de ruido coherente. Esto significa que el ruido no es totalmente aleatorio, como puede ser un ruido blanco, donde los elementos se generan de manera independiente los unos de los otros y no guardan relación entre sí.

En un ruido coherente los valores de los elementos se calculan de forma contextual; es decir, teniendo en cuenta otros elementos del conjunto. En generación procedural, generalmente se emplean algoritmos que tienen en cuenta los elementos que se encuentran alrededor.

En el caso de Diamond-Square, se emplean las cuatro esquinas de un cuadrado para calcular el centro, que es una media de los valores de los mismos más un desplazamiento aleatorio dentro de un rango concreto (esto se conoce como el paso del cuadrado); después, se emplea este centro y las esquinas para calcular los centros de los lados del cuadrado de la misma forma (de manera que se forma un diamante, de ahí que este paso se conozca como el paso del diamante).

Tras esto, se subdivide el cuadrado inicial en cuatro empleando estos puntos y se repite el proceso en cada uno de ellos, empleando un rango menor en los desplazamientos aleatorios en cada iteración. Así se va reduciendo la escala y deplazando el terreno de manera más detallada.

Debido a este proceso de repetición y reducción de escala (que de manera natural tiene una tendencia a la recursión), los terrenos generados por este tipo de algoritmos se conocen como "terrenos fractales.º "paisajes fractales".

4.3. Generación de agua

Como ya se ha explicado en el capítulo de diseño, la generación de agua está separada en dos módulos: el de generación de océanos y el de generación de ríos.

4.3.1. Generación de océanos

De nuevo, para la generación de océanos se ha elegido el algoritmo más eficiente y simple de implementar para el primer prototipo. En este caso, es una generación por altura muy sencilla: el usuario indica la altura del mar deseada y todo terreno que se

encuentre por debajo de esa altura se convierte en agua. Esto se hace recorriendo el conjunto de elementos que han sido generados por el algoritmo de ruido (de ahí que ese paso sea fundamental) y modificando el campo "material"de la estructura de datos cuando la altura del elemento es menor que el nivel del mar definido.

4.3.2. Generación de ríos

La generación de ríos es bastante más compleja, aunque también se ha intentado simplificar lo máximo posible para este prototipo. Por desgracia, han surgido problemas a causa de esta simplificación. Éstos se discutirán en el capítulo de resultados.

El enfoque que se ha dado a la generación de ríos está ligeramente basado en la progresión natural de un río en el sentido de que tiende a ir por el terreno con mayor caída posible por el efecto de la gravedad terrestre. Esto se consigue en la simulación utilizando una heurística que penalice caminos cuyo terreno desciende menos.

Los pasos para la generación de ríos son los siguientes:

1. Se generan semillas donde nacen los ríos. Éstas tienen mayor probabilidad de aparecer en una cierta franja de alturas debido al clima con mayores precipitaciones que habitualmente acompaña a dichas alturas. No existe el concepto de latitud en este prototipo, como se ha mencionado anteriormente, por lo que no se consideran climas relacionados con la latitud como los desiertos o los bosques tropicales.

   A alturas superiores a la de esta "franja de altas precipitaciones", se entiende que es posible que el agua caiga en forma de nieve y permanezca en estado sólido en el suelo, y a alturas inferiores no hay precipitaciones suficientes.

2. Para cada una de las semillas generadas de comienzo de río, se busca un final de río correspondiente. Este final se encuentra en la casilla de océano más próxima, con lo cual se realiza buscando en un cuadrado alrededor de la semilla y ampliando el lado del cuadrado hasta encontrarlo.

3. Mediante el algoritmo de *pathfinding* (búsqueda de caminos) A*, se busca un camino óptimo que comunica el inicio con el final del río, y se recorren cada una de las posiciones, cambiando el material del elemento correspondiente a agua (a no ser que ya fuera una posición de agua debido a que un río ha desembocado en otro).

   Este camino se busca empleando la altura de los elementos como coste en la función de coste del algoritmo de manera directamente proporcional; así, el algoritmo buscará siempre la mayor caída de terreno.

4.4. Renderizado

El renderizado se realiza sobre un elemento canvas que el usuario inserta en el HTML con el identificador "display". Su tamaño debe ser potencia de 2, con valores típicos de 256x256 ó 512x512. También es posible renderizar a tamaños con potencia de 2 para posteriormente estirar la imagen de manera que ocupe la ventana completa, pero esto queda en manos del usuario en esta primera versión.

También queda como tarea del usuario la colocación del elemento canvas dentro de la ventana, pues lo único que hace el proyecto es renderizar sobre él.

4.4.1. Renderizado en 2D

El renderizado topográfico se realiza recorriendo el array de elementos por orden y traduciendo la altura y material de cada elemento en un color determinado, para posteriormente dar color al pixel correspondiente en el canvas con las funciones de dibujo de javascript.

Debido a que el tamaño del canvas y el tamaño del terreno generado pueden diferir, pues son tamaños ambos determinados por el usuario, es necesario calcular el tamaño de cada pixel dentro del canvas. Se pueden dar varios casos:

1. **Que el tamaño del terreno sea menor que el canvas**. Por ejemplo, que el terreno generado sea de tamaño 256x256, mientras que el elemento canvas se haya creado con 512x512 pixels (por ejemplo, declarándolo en el documento HTML con los atributos width="512pxz height="512px"). En este caso, cada elemento del terreno ocupará más de un pixel del canvas, y a la hora de dibujarlo el rectángulo tendrá tamaño mayor que 1 (en concreto, su dimensión será de 2x2 pixels).

2. **Que el tamaño del terreno sea igual que el canvas**. Si ambos se han creado, por ejemplo, con tamaño 512x512, cada elemento del terreno corresponde a un pixel del canvas. El rectángulo dibujado será por tanto de tamaño 1x1 pixels.

3. **Que el tamaño del terreno sea mayor que el canvas**. Si el terreno se ha generado con tamaño 2048x2048 y el canvas es de 512x512 pixels, los elementos del terreno tendrán un tamaño menor que 1 al ser traducidos a un rectángulo para ser dibujados en el canvas.

El tamaño que tiene que tomar el rectángulo que se dibuja se calcula de forma sencilla

si se asume que el canvas creado es un cuadrado, simplemente dividiendo el tamaño del terreno generado entre la anchura o altura del elemento canvas.

Dibujar a tamaños menores de 1 pixel cuando el canvas no ha sido ampliado (es decir, que realmente se está intentando dibujar en niveles inferiores a 1 pixel) hará que el motor de renderizado del navegador agrupe los datos de varios elementos en 1, aunque esto dependerá de la plataforma. El resultado final será adecuado en cualquier caso, sin defectos visuales apreciables.

4.4.2. Renderizado en 3D

Como se ha comentado en el diseño del proyecto, para simplificar el renderizado en 3D se emplea la biblioteca three.js. La técnica de renderizado elegida ha sido la de nube de puntos, por la simpleza en su uso, y por el hecho de que no modifica el terreno, permitiendo así ver con claridad si la generación del terreno funciona correctamente. Conceptualmente es muy similar a las estructuras de datos con las que estamos trabajando.

Para crear y renderizar una escena en three.js, o de forma similar en la enorme mayoría de motores 3D, hay que establecer una serie de parámetros y crear y situar unos objetos en el espacio. En el Anexo A se incluyen los pasos a seguir para mostrar una nube de puntos en three.js.

1. Definir la geometría que vamos a renderizar. En nuestro caso, definir el conjunto de puntos sobre el espacio.

2. Para cada vértice de la geometría, asignar un color. En este prototipo, se usará el color del tipo de terreno al que pertenece el elemento asociado al vértice.

3. Crear un material para el objeto (la geometría). En este caso, three.js nos aporta un material específico para una nube de puntos del cual sólo tenemos que concretar algunos parámetros como el tamaño de cada punto, la opacidad...

4. Utilizando la geometría y el material, definimos la nube de puntos.

5. Ahora que está definida la nube, creamos una escena y la añadimos.

6. Definimos una cámara y la situamos en el espacio de manera que apunte hacia la escena con la distancia correcta. Una forma simple de hacerlo es dejarla apuntando hacia el origen (alrededor del cual estamos situando los elementos) y situarlo a una distancia proporcional al tamaño del terreno para poder siempre observarlo entero. Es interesante también rotarla en el eje X o Y respecto al origen, de manera que

siga apuntando hacia el centro de la imagen pero con una perspectiva picada, para poder observar la altura del terreno; de otra forma tendríamos una imagen 2D en la pantalla.

7. Crear el motor de render que queremos emplear; en este caso, un motor sobre WebGL.

8. Asignar al motor el elemento canvas de salida y el tamaño de renderizado que queremos.

9. Llamar a la función de renderizado del motor.

4.5. La página web

Como se ha comentado anteriormente, durante el desarrollo se han separado las funcionalidades en distintos archivos para facilitar su lectura; sin embargo, en javascript esto no tiene ninguna repercusión en la ejecución más allá de que en el HTML es necesario incluirlos todos. Por tanto, no tiene demasiada relevancia discutir cómo se ha separado el código entre los archivos, pues el código javascript se puede particionar como se desee entre archivos siempre que luego se incluyan todos los necesarios y no se generen errores de sintaxis.

En el capítulo de resultados se puede ver la interfaz que se desarrolló para poder ver los resultados de cada uno de los algoritmos y procesos y para poder jugar con los parámetros de los mismos.

4.6. Proceso de desarrollo

Partiendo del hecho de que el producto final va a estar basado en prototipos, y lo que se está desarrollando para este trabajo es el prototipo inicial de dicho producto, este prototipo inicial también se ha desarrollado siguiendo una especie de sub-prototipos.

El primer paso del desarrollo, elegido el diseño del prototipo y los módulos que incluirá, ha sido la implementación del algoritmo de generación de ruido. Para ello se ha recurrido a diversas fuentes donde se explicaba el funcionamiento del mismo y se ha procedido a implementarlo.

Una vez implementado el algoritmo, es necesario visualizar los resultados para comprobar que se ha logrado obtener el ruido esperado. Por ello, el segundo paso del

desarrollo ha sido implementar una forma simple de visualizar dicho ruido. Esto se hace simplemente recorriendo los elementos e imprimiendo sobre el canvas un cuadrado por elemento, donde el valor rgb del mismo viene dado por la altura del elemento, teniendo las tres componentes del rgb el mismo valor.

Así conseguimos una escala de grises, como se ha mencionado anteriormente. En la figura 4.1 podemos observar los resultados.

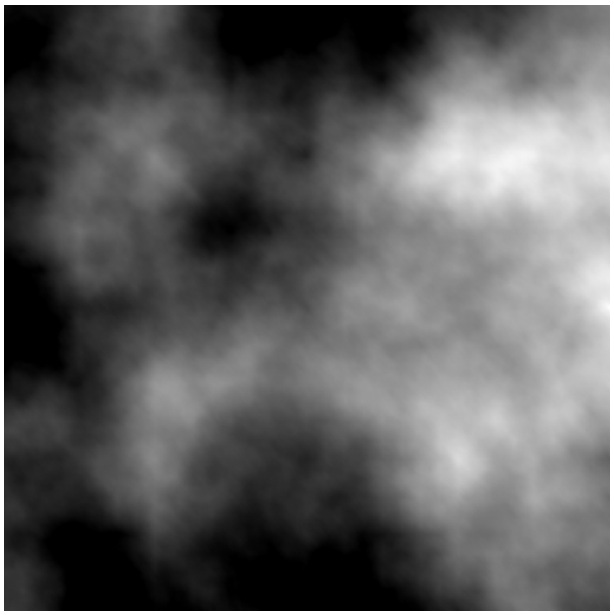

Figura 4.1: Visualización del ruido generado

Esta funcionalidad de renderizado en escala de grises se ha mantenido y encapsulado en una función de manera que pueda seguir siendo utilizada en el futuro para probar las modificaciones que se le hagan al algoritmo, o para probar otros algoritmos de generación de ruido.

El tercer paso fue la generación de agua. La generación de océanos no presentó ninguna dificultad con el algoritmo que se ha elegido. Como ya se ha comentado, sólo se recorren los elementos y se cambia el material a agua de todos aquellos cuya altura sea menor que la del nivel del mar.

La generación de ríos sí ha sido uno de los aspectos más laboriosos del proyecto, pues se barajaron muchos enfoques y algoritmos hasta dar con uno lo suficientemente simple y convincente. El finalmente empleado satisface los resultados esperados, pero no está por ello libre de problemas, como se ha comentado. En el capítulo de resultados se discute esto más en profundidad.

Originalmente la propuesta de generación de ríos buscaba simplemente el vecino más

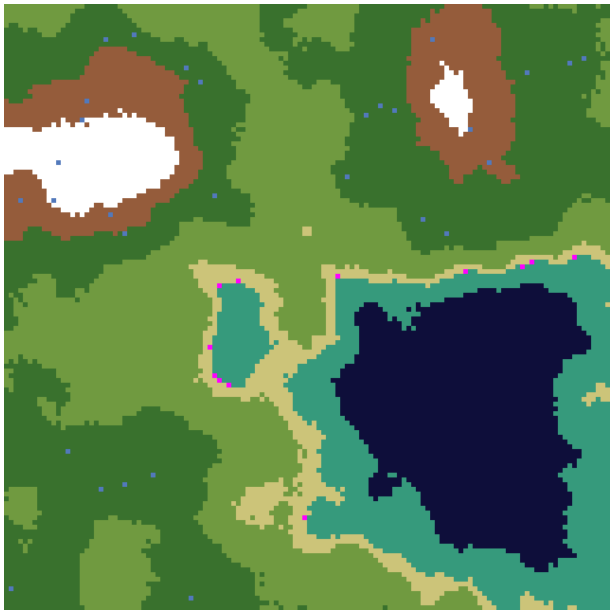

Figura 4.2: Comienzos y finales de los ríos creados

bajo en cada posición, y si no existía ningún vecino inferior, se creaba un pequeño lago subiendo el nivel del agua y se proseguía a buscar si existía algún vecino inferior durante un par de iteraciones más.

Este algoritmo, aunque conceptualmente mucho más sencillo que el finalmente implementado, es en la práctica mucho más complejo si no se emplea la orientación correcta donde las posiciones de agua pertenezcan a un río en concreto para nunca repetir, lo cual requiere un sistema de clases bastante más aparatoso.

Además, el sistema de que un río puede generar un lago y éste a su vez puede generar otro río implica que el algoritmo tiende a la recursión, e incluye mucho "hard-coding"de búsqueda de posiciones. Todo esto en definitiva hacía que esta propuesta implicara un proceso mucho más tedioso de programación, en un lenguaje además poco preparado para este tipo de trabajos.

Por ello se eligió la propuesta mucho más limpia de búsqueda del océano más cercano con A*.

Para poder visualizar los diferentes materiales es necesaria otra técnica de renderizado que la escala de grises, que sólo permite observar la altura del elemento. Con los tipos de terreno definidos, la implementación del renderizado topográfico fue sencilla.

Se designaron unas alturas para cada uno de los tipos de terrenos, de tal forma que se crearon franjas dentro de las cuales todos los elementos tenían el mismo color, y ese color era el que se aplicaba al cuadrado que representaba a cada elemento. Era necesario

sin embargo comprobar el material del elemento para alturas superiores al nivel del mar, pues si éste era agua, el color correcto es el de río.

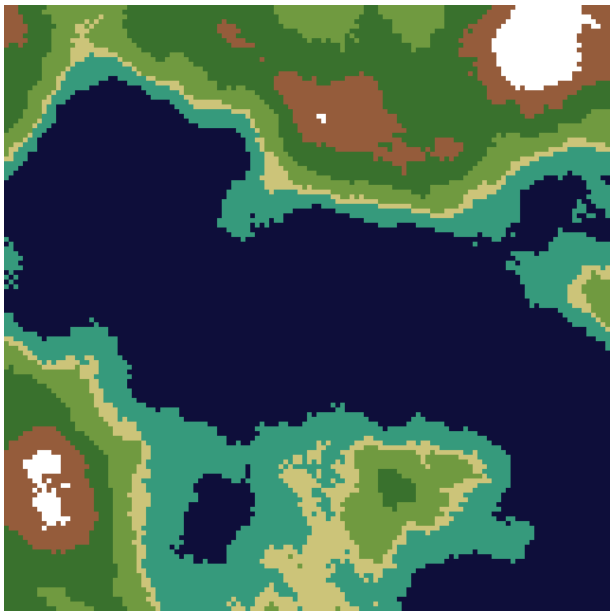

Figura 4.3: Renderizado topográfico de un mapa de 128x128

Así se conseguían las curvas de nivel de un mapa topográfico, como se ha comentado antes. En la figura 4.3 podemos observar un renderizado de un mapa de 128x128.

Una vez que se corrigieron todos los errores y problemas que surgieron y se vio que la generación de ruido y agua era correcta, se procedió a la implementación del renderizado 3D. Existen numerosos ejemplos de uso en la página web de three.js que, junto a la documentación de desarrollo, hicieron bastante fácil crear un ejemplo de renderizado de point cloud. El paso complicado fue definir la geometría y los materiales como necesita three.js utilizando los valores de los elementos.

Otra de las ventajas de emplear el algoritmo de generación de ruido Diamond-Square es que para cualquier tamaño se observan figuras reconocibles al renderizar; es decir, en un terreno de 64x64 ya se pueden apreciar montañas y océanos como para entender la forma del mapa.

Eso sí, hay un tamaño mínimo antes de que se pueda formar cualquier cosa, alrededor del 32x32, puesto que para tamaños menores no hay espacio para generarse ninguna formación. En las figuras 4.4, 4.5 y 4.6 se muestran algunos tamaños significativos que sirven, junto a la anteriormente mencionada figura 4.3 para observar la progresión.

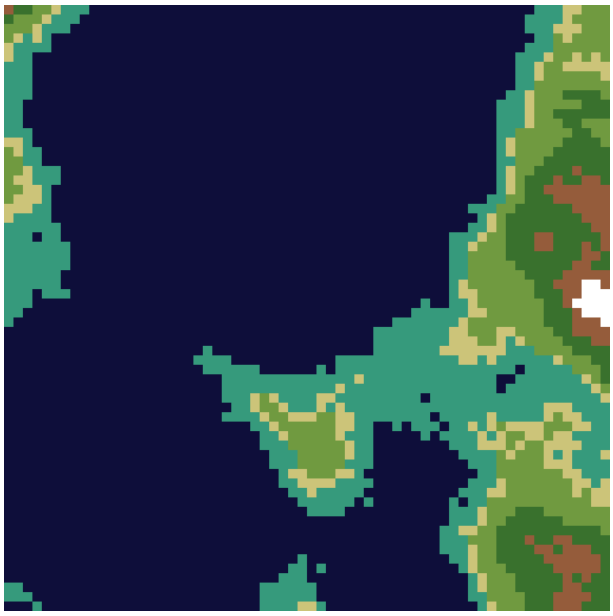

Figura 4.4: Renderizado topográfico de un mapa de 64x64

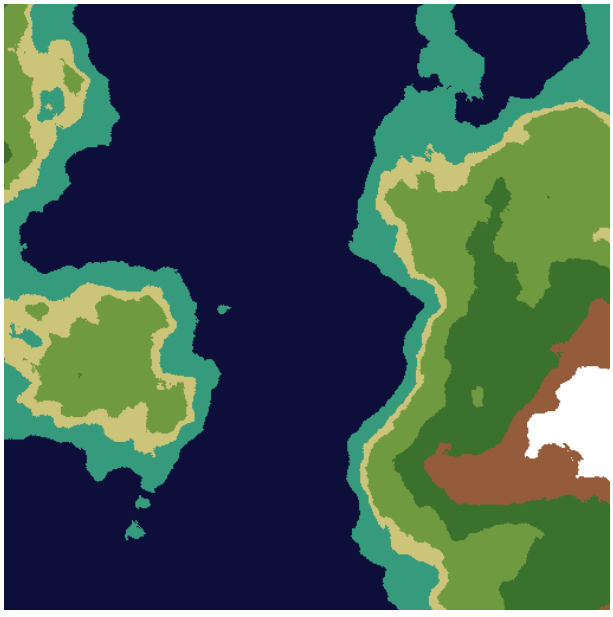

Figura 4.5: Renderizado topográfico de un mapa de 512x512

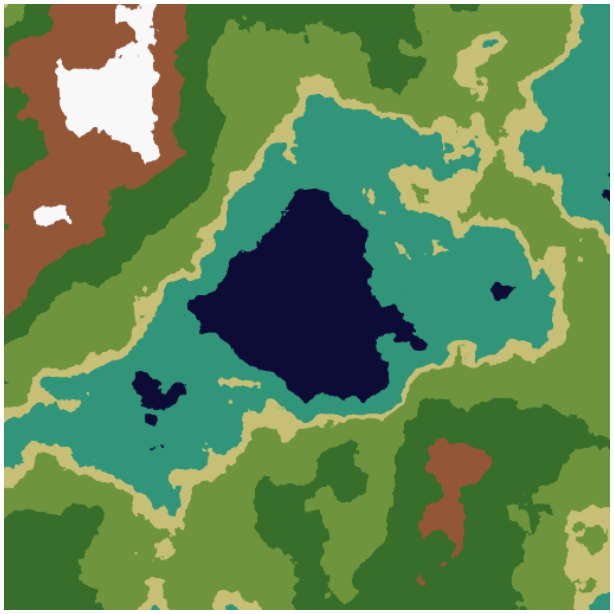

Figura 4.6: Renderizado topográfico de un mapa de 2048x2048

5

Resultados

Durante la fase de desarrollo, se ha preparado una plataforma web para la verificación de resultados y pruebas de los módulos implementados.

Esta plataforma consiste en un servidor python con una serie de archivos HTML que permiten acceder a las funcionalidades de la biblioteca de distinta forma y modificar parámetros para observar cómo afectan al resultado final.

De forma básica, se han escrito los siguientes tres archivos HTML:

- **create_map.html** Genera la estructura de datos de un terreno y la imprime por pantalla usando el siguiente formato: Nmaterial,datomaterial,dato..., donde N es el tamaño del terreno, medido en el número de filas y los conjuntos material,dato son cada uno de los elementos del terreno en orden (ordenados por filas). material puede ser "ground.º "waterz dato es un decimal que representa la altura del elemento.
- **load_map.html** Toma una serie de datos utilizando el formato anteriormente mencionado y los renderiza en pantalla. Este modo, junto al anterior, es útil para guardar un mapa generado en un archivo del equipo y cargarlo múltiples veces cuando se están haciendo pruebas sobre el renderizado.
- **create_load_map.html** Genera un nuevo mapa y a continuación lo renderiza,

combinando ambos modos anteriores pero sin mostrar en ningún momento la estructura de datos. Este modo es útil para hacer pruebas sobre los algoritmos de generación, pues se pueden generar y visualizar terrenos rápidamente.

Para la gran mayoría de pruebas a lo largo del desarrollo se ha utilizado la interfaz de generación y renderizado de terrenos (create_load_map.html), pues empleando terrenos no muy grandes (como mucho de 512x512) la generación del terreno es tan rápida que no merece la pena utilizar las interfaces de generación y renderizado por separado, por el tiempo que lleva cambiar de una a otra y, principalmente, por el tiempo que lleva al sistema operativo copiar en el portapapeles toda la información del terreno, y volverlo a pegar. Además, cabe destacar que también tarda significantemente más tiempo el navegador en imprimir los datos en la página que en renderizarlos en el canvas.

En su primera versión, debido a que cada módulo se ha implementado utilizando sólo un algoritmo y no existen alternativas para el usuario (excepto para el renderizado), no hay muchos parámetros que modificar a la hora de realizar pruebas aún. Se han usado los siguientes durante las pruebas:

- Tamaño del terreno. Debe ser potencia de 2.
- Rugosidad del terreno. Este parámetro afecta a los desplazamientos aleatorios que se hacen al calcular un elemento.
- Módulo de agua
  - Generación de océanos. Permite activar o desactivar la generación de océanos.
  - Nivel del mar.
  - Generación de ríos. Permite activar o desactivar la generación de ríos.
  - Frecuencia de aparición de ríos. Este número entre 0 y 1 indica la probabilidad que tiene de aparecer un río en la franja de altas precipitaciones. Fuera de esa franja, este valor disminuye progresivamente.
- Algoritmo de renderizado a elegir.

En la figura 5.1 se puede ver una captura de pantalla de la interfaz que se usaba durante el desarrollo para la realización de pruebas.

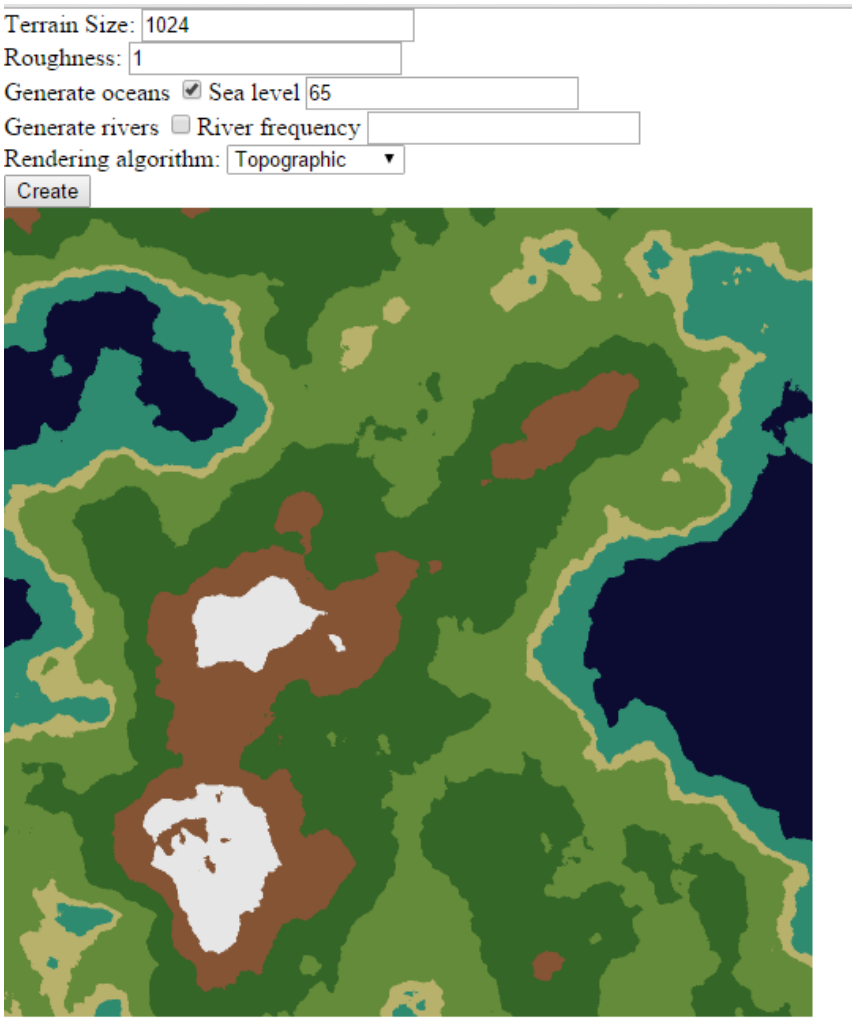

Figura 5.1: Interfaz de uso de la página de prueba de generación y renderizado

5.1. Rendimiento

Mediante funciones de obtención de tiempo (getTime antes y después de la ejecución de los algoritmos), se han calculado los tiempos de ejecución de la generación para distintos tamaños.

La tabla 5.1 muestra los tiempos de ejecución con todos los módulos activos (generación de ruido, generación de océanos y ríos y renderizado topográfico):

Tiempos		
Tamaño	Número de elementos	Tiempo(ms)
32x32	1024	18
64x64	4096	56
128x128	16384	194
256x256	65536	312
512x512	262144	841
1024x1024	1048576	7088
2048x2048	4194304	581894
4096x4096	16777216	memoria insuficiente

Tabla 5.1: Tiempos de ejecución para distintos tamaños de terreno

5.2. Generación de ríos

A continuación vamos a comentar algunos de los problemas encontrados relacionados con el enfoque que se ha dado a la generación de ríos.

Este paso ha sido uno de los más complejos del desarrollo, debido a la multitud de formas que existen de simular e implementar ríos naturales.

La solución final ha sido elegida, como se ha comentado en varias ocasiones, por su simplicidad a la hora de ser implementada. Es totalmente posible que existan soluciones más eficientes, más realistas o más simples que la elegida, pero ésta se considera ampliamente adecuada para el alcance de este trabajo.

Apostar por la implementación más simple no viene libre de problemas. A continuación se detallan algunos aspectos problemáticos:

Dado que uno de los primeros pasos en la generación de ríos es encontrar el océano más cercano, la generación de ríos no es posible si no se han generado océanos. Esto implica dos cosas:

1. El usuario no puede elegir no generar océanos para después elegir generar ríos. Esto resulta imposible con el enfoque actual, y por ello se para la ejecución y se notifica al usuario del problema.

2. La generación del mapa de alturas podría crear un terreno en el que todos los elementos están por encima del nivel del mar, bien porque el nivel del mar elegido es demasiado bajo o bien por un caso puntual debido a la aleatoriedad intrínseca al algoritmo. Esto impediría la generación de ríos, por lo que, en caso de que el usuario haya elegido generarlos, se salta este paso y se notifica al usuario de la razón.

Otro de los problemas del paso de elegir el final del río como la casilla más cercana de océano y hacer que el río tenga que desembocar allí es que en muchos casos esto resulta antinatural, puesto que puede obligar al río a ascender. Hay dos casos esenciales a discutir:

- **La casilla de nacimiento del río se encuentra en un lado del mapa y el océano en otro**. En este caso lo más probable es que entre el nacimiento del río y el océano se encuentre una montaña, partiendo del hecho de que los ríos se generan con mucha mayor probabilidad en terrenos altos.

  La solución que parecería más natural sería que el río fuera hacia el borde más cercano, dando a entender que en alguna posición que no vemos hay otro océano; sin embargo, como esto no se contempla, A* llevará el río hasta el océano aunque eso signifique atravesar una montaña.

- **El océano más cercano es mucho más inaccesible que uno un poco más lejano**. Como se parte de la posición más cercana y no se investiga más allá, es posible en muchos casos que ésta no sea la más natural y, de nuevo, el algoritmo obligue al río a tomar rutas que resultan poco creíbles.

Un pequeño problema con la implementación actual que hace perder credibilidad a los ríos en muchas ocasiones es el hecho de que se busca siempre el camino óptimo entre el origen y el final del río. Esto es una buena técnica para la generación de caminos y carreteras, puesto que como se construyen conociendo el contexto y usando una cierta inteligencia, tienden a ser caminos óptimos; sin embargo, en el caso de los ríos, rara vez esto es así, sobre todo si se supone el paso del tiempo (aunque en nuestro proyecto este concepto no exista).

Hablando de la ausencia del paso del tiempo, la falta de simulación de erosión a causa del curso del agua hace que los ríos parezcan mucho más artificiales. No se ha planteado implementar esto, pues un módulo de simulación de erosión es una tarea de envergadura tan grande como todo este trabajo.

Por último, cabe comentar que el algoritmo no aumenta el caudal de un río cuando éste posee afluentes; de hecho, no existe el concepto de afluente. Cuando dos ríos coinciden en un largo tramo, el segundo que llega se deja de simular puesto que el camino óptimo ya se ha encontrado. Si se aumentara el caudal del mismo se generarían ríos mucho más interesantes.

6

Conclusiones

Al estar el trabajo dividido entre los algoritmos de generación procedural de terrenos y el renderizado de dichos terrenos generados, se han quedado muchos cabos sueltos en ambos aspectos. El trabajo debería ser expandido principalmente en el área de la generación (ya que era el tema principal del mismo) aunque se haya visto retrasada debido al gran trabajo que lleva la preparación de un motor de renderizado (en concreto, a adquirir los conocimientos necesarios para entender cómo funciona dicho motor y cómo implementarlo, aunque finalmente se haya usado una herramienta externa).

El principal aspecto a mejorar en el generador es la adición de mayores elementos geográficos, geológicos y climatológicos: lagos, caídas de agua, vegetación (distintos bosques, tundra, desierto, selva...), cuevas, etc. También sería interesante generar elementos de creación humana, como ciudades, pueblos y carreteras. La generación de carreteras es similar a la generación de ríos, y daría mejores resultados al estar buscando siempre caminos óptimos entre dos puntos, pues las carreteras se suelen construir de manera ïnteligente", como se ha comentado anteriormente.

También es importante pulir ciertos algoritmos desarrollados, en cuanto al rendimiento y en cuanto a sus capacidades para producir terrenos realistas; así como aportar más formas al usuario de generar los mismos elementos (distintos algoritmos de generación de ruido, de generación de océanos...).

Una capacidad que sería interesante implementar es la de simular erosión y, en general, el paso del tiempo. Algunos videojuegos se basan en estas técnicas para crear un mundo rico donde las civilizaciones se crean y mueren, dejando tras de sí una historia en forma de restos de poblados o monumentos, y donde el mundo se ve afectado por la erosión y la acción humana. Un gran ejemplo es el videojuego *Dwarf Fortress*, desarrollado por dos hermanos (aún en desarrollo) y considerado por muchos el mayor RPG jamás creado. Esto por supuesto no es trivial y constituye una cantidad de trabajo posiblemente exponencialmente mayor a la dedicada a la simple generación de terreno que se ha hecho en este trabajo.

Bibliografía

Libros

1. Anyuru, Andreas (2012), *Professional WebGL programming developing 3D graphics for the web*, Wrox

Recursos en la web

1. *three.js - Javascript 3D Library.* `http://threejs.org/`
2. Alvarez-Buylla, Ebyan. "World Generation Breakdown". `http://www.nolithius.com/game-development/world-generation-breakdown`
3. Cooter, Kaelan. Randomly generated world map". `http://blog.kaelan.org/randomly-generated-world-map/`
4. Patel, Amit. "Polygonal Map Generation for Games". `http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/`
5. Ramires Fernandes, António. "Terrain Tutorial - Mid Point Displacement Algorithm". `http://www.lighthouse3d.com/opengl/terrain/index.php?mpd2`
6. Martz, Paul. "Generating Random Fractal Terrain". `http://www.gameprogrammer.com/fractal.html`
7. Black, Davian. Random River Generation". `http://bigbadwofl.blogspot.com.es/2013/02/random-river-generation.html`

Apéndices

A

Anexos

A.1. Anexo A: ejemplo de nube de puntos en three.js

A continuación podemos ver extracto de código de ejemplo que renderiza una esfera como nube de puntos en three.js [1] [2].

[1] En el siguiente enlace se puede ver el código original en acción. http://jsfiddle.net/J7zp4/88/

[2] Hilo de stackoverflow donde se publicó dicho código: http://stackoverflow.com/questions/12659461/rendering-a-large-number-of-colored-particles-using-three-js-and-the-canvas-rend

```
var geometry = new THREE.SphereGeometry( 100, 32, 16 );

var colors = [];
for( var i = 0; i < geometry.vertices.length; i++ ) {
        colors[i] = new THREE.Color();
        colors[i].setHSL( Math.random(), 1.0, 0.5 );

}
geometry.colors = colors;

material = new THREE.PointCloudMaterial( {
        size: 10,
        transparent: true,
        opacity: 0.7,
        vertexColors: THREE.VertexColors
} );

pointCloud = new THREE.PointCloud( geometry, material );

var scene = new THREE.Scene();
scene.add( pointCloud );

var camera = new THREE.PerspectiveCamera( 75, window.
   ↪ innerWidth/window.innerHeight, 0.1, 1000 );
camera.position.z = terrainSize;

var canvas = document.getElementById("display");
var renderer = new THREE.WebGLRenderer( { canvas: canvas })
   ↪ ;
renderer.setSize( canvas.width, canvas.height );

renderer.render( scene, camera );
```

A.2. Anexo B: Manual del programador

La biblioteca desarrollada se compone de un único archivo de código javascript que ha de ser incluido en el archivo HTML sobre el que se está trabajando.

Los pasos a seguir para usar la biblioteca son los siguientes:

1. Dar valor a la variable terrainSize con el tamaño del terreno a generar. Éste debe ser una potencia de 2.

2. Dar valor a la variable roughnessConstant si se desea variar la rugosidad del terreno. El valor por defecto es 1.

3. Llamar a la función de generación de ruido diamondSquare(). Los datos del terreno se almacenarán en una variable llamada terrain.

4. Si se desea, establecer el nivel del mar y llamar a la función de generación de océanos: generateOceans(). Esta función presupone que se ha llamado antes a la función de generación de ruido, pues utiliza y modifica los datos de la variable terrain.

5. Si se desea, establecer la frecuencia de aparicióin de ríos y llamar a la función de generación de ríos: generateRivers(). Esta función requiere que se ha llame antes a la función de generación de océanos; en caso contrario, la ejecución se detendrá y se mostrará un mensaje de error. Llamar a esta función modifica nuevamente los datos de la variable terrain.

6. Llamar, si se desea, a alguna de las funciones de renderizado que acompañanan a la biblioteca: renderNoise(), renderTopographic() o renderPointCloud(). Las tres funciones presuponen la existencia de un elemento canvas con el identificador "display", de altura y anchura iguales y siendo una potencia de 2.